universität innsbruck

Functional Programming	WS 2019/2020	LVA 703024+703025
Exercises 5		Deadline: November 6, 2019, 6am

- Please write all your Haskell functions from this exercise sheet into a single .hs-file and upload it in OLAT.
- You can use a template .hs-file that is provided on the proseminar page.
- The file should compile with ghci.
- Once the file has been uploaded, it cannot be changed or resubmitted!

Exercise 5.1 *Type Classes* **3 p.**

This exercise should show you how types and type classes can be used to model data. Consider the following data types and the following type class:

```
data City = Innsbruck | Munich | Graz | Gramais
data Country = Austria | Germany
data State = Tyrol | Bavaria | Styria

class Info a where
  population :: a -> Integer
  area :: a -> Integer
```

1. Make the types `City`, `Country` and `State` instances of the class `Info`, cf. slides 15 – 16 of part 3, using the following data: [1]

Location	Type	Population	Area in km^2
Innsbruck	City	132110	105
Munich	City	1471508	105
Graz	City	288806	127
Gramais	City	41	32
Bavaria	State	13076721	70550
Tyrol	State	754705	12640
Styria	State	1243052	16401
Germany	Country	83019213	357578
Austria	Country	8858775	83878

(1 point)

2. Define the following functions:

```
inhabitantsPerSkm :: (Info a) => a -> Integer
hasMoreArea :: (Info a, Info b) => a -> b -> Bool
hasMoreInhabitants :: (Info a, Info b) => a -> b -> Bool
```

The function `inhabitantsPerSkm` should return the number of inhabitants per km^2 rounded down. `hasMoreArea x y` should return `True` if `x` has a larger area then `y`, otherwise it should return `False`. `hasMoreInhabitants x y` should return `True` if `x` has more inhabitants then `y`, otherwise it should return `False`.

[1] Data taken from `https://de.wikipedia.org/`

Note that the notation `(Info a, Info b) =>` means that `a` may only be instantiated by a type that is an instance of type-class `Info`, and additionally `b` can only be instantiated by instances of `Info`. (1 point)

3. Make the types `Country` and `State` instances of the following type class:

```
class Located a where
  isIn :: City -> a -> Bool
```

`isIn x y` should return `True` if city `x` is in state or country `y`, otherwise `False`. For example, `isIn Gramais Tyrol` should evaluate to `True`. (1 point)

Exercise 5.2 *Pattern Matching* **3 p.**

```
fun_1 True True   = False
fun_1 True False  = False
fun_1 False True  = True
fun_1 False False = False

fun_2 True  _     = False
fun_2 False y     = y

fun_3 x     True  = not x
fun_3 _     False = False

fun_4 True  True  = False
fun_4 False True  = True
fun_4 _     False = False
```

We say that a binary Haskell function f is equal to g w.r.t. a set of inputs I, if and only if for all inputs $x \in I$ and $y \in I$ the equality $f\ x\ y = g\ x\ y$ is satisfied.

1. Consider the Haskell programs above. Which of the functions `fun_2`,...,`fun_4` are equal to `fun_1` w.r.t. inputs {`True`, `False`}? For each inequality, provide some input which distinguishes the functions. (1 point)
2. Which of the functions `fun_2`,...,`fun_4` are equal to `fun_1` w.r.t. inputs {`True`, `False`, $\perp$}? (As usual, $\perp$ is a special value that represents non-termination or abnormal termination via errors.) For each inequality, provide some input which distinguishes the functions. You should solve this part without invoking GHCi, but instead understand pattern matching as it is explained on slides 10 – 11 in part 3. (2 points)

Exercise 5.3 *Approximations with Floating Point Numbers* **4 p.**

The number $\pi = 3.14159265358979323846264 33\ldots$ is the ratio of a circle's circumference to its diameter. A variation of Leibniz formula to define π is the following one:

$$\pi = \sum_{n=0}^{\infty} a_n,$$

where

$$a_n = \frac{8}{(4 \cdot n + 1)(4 \cdot n + 3)}.$$

Pruning the infinite sum at some point we obtain π_k, defined as

$$\pi_k = \sum_{n=0}^{k} a_n,$$

which is an approximation of π.

1. Write a polymorphic function that defines π_k by computing the sum from left-to-right, i.e.,

$$(\ldots((a_0 + a_1) + a_2)\ldots) + a_k.$$

Of course you may define further auxiliary functions which can be reused in the whole exercise. You should have a detailed look at the `Num` type-class, e.g., slide 19 of part 3 or at the following URL:

`http://hackage.haskell.org/package/base-4.12.0.0/docs/Prelude.html#g:7`

(1 point)

2. Write a polymorphic function that defines π_k by computing the sum from right-to-left, i.e.,

$$a_0 + (a_1 + \ldots (a_{k-1} + a_k)) .$$

(1 point)

3. Create a table where you write down the values that you get for $k = 1, 100, 10000$ and 1000000 for both function definitions using both `Float` and `Double`. Try to interpret your results. (1 point)

4. Since both `Float` and `Double` can only present finitely many different numbers, and since π_k is increasing in k there must be some k such that π_k `==` π_{k+1} `==` π_{k+2} `==` $\ldots$ where here `==` refers to Haskell's equality-operation on floating point numbers. Such a k is clearly optimal in the sense that no further progress can be made.

   Write a polymorphic constant of type `(Eq a, Fractional a) => a` that defines this *best possible approximation of* π *based on* π_k, i.e., one which stops as soon as π_k `==` π_{k+1}. Note that the notation `(Eq a, Fractional a) =>` means that the type-variable `a` may only be instantiated by types which instantiate both type-classes, `Eq` and `Fractional`.

   The basic algorithm (testing π_0 `==` π_1, then π_1 `==` π_2 until one finds some k such that π_k `==` π_{k+1}) is too inefficient if defined naively, since then in every iteration an approximation of π is recomputed from scratch. Try to develop a more efficient version, e.g., by reusing ideas of the efficient Fibonacci algorithm.

   What is the result for `Float`? Compare it to the `Float`-results of part 3 and explain. (1 point)